# Linux 进程与线程的区别

进程与线程的区别，早已经成为了经典问题。自线程概念诞生起，关于这个问题的讨论就没有停止过。无论是初级程序员，还是资深专家，都应该考虑过这个问题，只是层次角度不同罢了。一般程序员而言，搞清楚二者的概念，在工作实际中去运用成为了焦点。而资深工程师则在考虑系统层面如何实现两种技术及其各自的性能和实现代价。以至于到今天，Linux 内核还在持续更新完善(关于进程和线程的实现模块也是内核完善的任务之一)。

本文将以一个从事 Linux 平台系统开发的程序员角度描述这个经典问题。本文素材全部来源于工作实践经验与知识规整，若有疏漏或不正之处，敬请读者慷慨指出。

0.首先，简要了解一下进程和线程。对于操作系统而言，进程是核心之核心，整个现代操作系统的根本，就是以进程为单位在执行任务。系统的管理架构也是基于进程层面的。在按下电源键之后，计算机就开始了复杂的启动过程，此处有一个经典问题：当按下电源键之后，计算机如何把自己由静止启动起来的？本文不讨论系统启动过程，请读者自行科普。操作系统启动的过程简直可以描述为上帝创造万物的过程，期初没有世界，但是有上帝，是上帝创造了世界，之后创造了万物，然后再创造了人，然后塑造了人的七情六欲，再然后人类社会开始遵循自然规律繁衍生息。操作系统启动进程的阶段就相当于上帝造人的阶段。本文讨论的全部内容都是“上帝造人”之后的事情。第一个被创造出来的进程是 0 号进程，这个进程在操作系统层面是不可见的，但它存在着。0 号进程完成了操作系统的功能加载与初期设定，然后它创造了 1 号进程（init），这个 1 号进程就是操作系统的“耶稣”。1 号进程是上帝派来管理整个操作系统的，所以在用 pstree 查看进程树可知，1 号进程位于树根。再之后，系统的很多管理程序都以进程身份被 1 号进程创造出来，还创造了与人类沟通的桥梁——shell。从那之后，人类可以跟操作系统进行交流，可以编写程序，可以执行任务。

而这一切，都是基于进程的。每一个任务(进程)被创建时，系统会为他分配存储空间等必要资源，然后在内核管理区为该进程创建管理节点，以便后来控制和调度该任务的执行。进程真正进入执行阶段，还需要获得 CPU 的使用权，这一切都是操作系统掌管着，也就是所谓的调度，在各种条件满足(资源与 CPU 使用权均获得)的情况下，启动进程的执行过程。除 CPU 而外，一个很重要的资源就是存储器了，系统会为每个进程分配独有的存储空间，当然包括它特别需要的别的资源，比如写入时外部设备是可使用状态等等。有了上面的引入，

我们可以对进程做一个简要的总结：

进程，是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配的基本单位，是操作系统结构的基础。它的执行需要系统分配资源创建实体之后，才能进行。

随着技术发展，在执行一些细小任务时，本身无需分配单独资源时(多个任务共享同一组资源即可，比如所有子进程共享父进程的资源)，进程的实现机制依然会繁琐的将资源分割，这样造成浪费，而且还消耗时间。后来就有了专门的多任务技术被创造出来——线程。线程的特点就是在不需要独立资源的情况下就可以运行。如此一来会极大节省资源开销，以及处理时间。

1.好了，前面的一段文字是简要引入两个名词，即进程和线程。本文讨论目标是解释清楚进程和线程的区别，关于二者的技术实现，请读者查阅相关资料。

下面我们开始重点讨论本文核心了。从下面几个方面阐述进程和线程的区别。

1).二者的相同点

2).实现方式的差异

3).多任务程序设计模式的区别

4).实体间(进程间，线程间，进线程间)通信方式的不同

5).控制方式的异同

6).资源管理方式的异同

7).个体间辈分关系的迥异

8).进程池与线程池的技术实现差别

接下来我们就逐个进行解释。

1).二者的相同点

无论是进程还是线程，对于程序员而言，都是用来实现多任务并发的技术手段。二者都可以独立调度，因此在多任务环境下，功能上并无差异。并且二者都具有各自的实体，是系统独立管理的对象个体。所以在系统层面，都可以通过技术手段实现二者的控制。而且二者所具有的状态都非常相似。而且，在多任务程序中，子进程(子线程)的调度一般与父进程(父线程)平等竞争。其实在 Linux 内核 2.4 版以前，线程的实现和管理方式就是完全按照进程方式实现的。在 2.6 版内核以后才有了单独的线程实现。

2).实现方式的差异

进程是资源分配的基本单位，线程是调度的基本单位。这句经典名言已流传数十年，各种操作系统教材都可见此描述。确实如此，这就是二者的显著区别。读者请注意“基本”二字。相信有读者看到前半句的时候就在心里思考，“进程岂不是不能调度？”，非也！进程和线程都可以被调度，否则多进程程序该如何运行呢！只是，线程是更小的可以调度的单位，也就是说，只要达到线程的水平就可以被调度了，进程自然可以被调度。它强调的是分配资源时的对象必须是进程，不会给一个线程单独分配系统管理的资源。若要运行一个任务，想要获得资源，最起码得有进程，其他子任务可以以线程身份运行，资源共享就行了。

简而言之，进程的个体间是完全独立的，而线程间是彼此依存的。多进程环境中，任何一个进程的终止，不会影响到其他进程。而多线程环境中，父线程终止，全部子线程被迫终止(没有了资源)。而任何一个子线程终止一般不会影响其他线程，除非子线程执行了 exit() 系统调用。任何一个子线程执行 exit()，全部线程同时灭亡。

其实，也没有人写出只有线程而没有进程的程序。多线程程序中至少有一个主线程，而这个主线程其实就是有 main 函数的进程。它是整个程序的进程，所有线程都是它的子线程。我们通常把具有多线程的主进程称之为主线程。

从系统实现角度讲，进程的实现是调用 fork 系统调用：

```
pid_t fork(void);
```

线程的实现是调用 clone 系统调用：

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg, ...
/* pid_t *ptid, struct user_desc *tls, pid_t *ctid */
);
```

其中，fork()是将父进程的全部资源复制给了子进程。而线程的 clone 只是复制了一小部分必要的资源。在调用 clone 时可以通过参数控制要复制的对象。可以说，fork 实现的是 clone 的加强完整版。当然，后来操作系统还进一步优化 fork 实现——写时复制技术。在子进程需要复制资源(比如子进程执行写入动作更改父进程内存空间)时才复制，否则创建子进程时先不复制。

实际中，编写多进程程序时采用 fork 创建子进程实体。而创建线程时并不采用 clone 系统调用，而是采用线程库函数，而应用最为广泛的是 POSIX 线程库。因此读者在多线程程序中看到的是 pthread_create 而非 clone。我们知道，库是建立在操作系统层面上的功能集合，因而它的功能都是操作系统提供的。由此可知，线程库的内部很可能调用了 clone。不管是

进程还是线程的实体，都是操作系统上运行的实体。

最后，我们说一下 vfork() 。这也是一个系统调用，用来创建一个新的进程。它创建的进程并不复制父进程的资源空间，而是共享，也就说实际上 vfork 实现的是一个接近线程的实体，只是以进程方式来管理它。并且，vfork()的子进程与父进程的运行时间是确定的：子进程“结束”后父进程才运行。请读者注意“结束”二字。并非子进程完成退出之意，而是子进程返回时。一般采用 vfork()的子进程，都会紧接着执行 execv 启动一个全新的进程，该进程的进程空间与父进程完全独立不相干，所以不需要复制父进程资源空间。此时，execv 返回时父进程就认为子进程“结束”了，自己开始运行。实际上子进程继续在一个完全独立的空间运行着。举个例子，比如在一个聊天程序中，弹出了一个视频播放器。你说视频播放器要继承你的聊天程序的进程空间的资源干嘛？莫非视频播放器想要窥探你的聊天隐私不成？

3).多任务程序设计模式的区别

由于进程间是独立的，所以在设计多进程程序时，需要做到资源独立管理时就有了天然优势，而线程就显得麻烦多了。比如多任务的 TCP 程序的服务端，父进程执行 accept()一个客户端连接请求之后会返回一个新建立的套接字描述符 scoketfd，此时如果 fork()一个子进程，将 scoketfd 带入到子进程空间去处理该连接的请求，父进程继续 accept 等待别的客户端连接请求，这样设计非常简练，而且父进程可以用同一变量(val)保存 accept()的返回值，因为子进程会复制 val 到自己空间，父进程再覆盖此前的值不影响子进程工作。但是如果换成多线程，父线程就不能复用一个变量 val 多次执行 accept()了。因为子线程没有复制 val 的存储空间，而是使用父线程的，如果子线程在读取 val 时父线程接受了另一个客户端请求覆盖了该值，则子线程无法继续处理上一次的连接任务了。改进的办法是子线程立马复制 val 的值在自己的栈区，但父线程必须保证子线程复制动作完成之后再执行新的 accept()。但这执行起来并不简单，因为子线程与父线程的调度是独立的，父线程无法知道子线程何时复制完毕。这又得发生线程间通信，子线程复制完成后主动通知父线程。这样一来父线程的处理动作必然不能连贯，比起多进程环境，父线程显得效率有所下降。

PS：这里引述一个知名的面试问题：多进程的 TCP 服务端，能否互换 fork()与 accept()的位置？请读者自行思考。

关于资源不独立，看似是个缺点，但在有的情况下就成了优点。多进程环境间完全独立，要实现通信的话就得采用进程间的通信方式，它们通常都是耗时间的。而线程则不用任何手段数据就是共享的。当然多个子线程在同时执行写入操作时需要实现互斥，否则数据就写“脏”

了。

4).实体间(进程间，线程间，进线程间)通信方式的不同

进程间的通信方式有这样几种：

A.共享内存　B.消息队列　C.信号量　D.有名管道　E.无名管道　F.信号

G.文件　　H.socket

线程间的通信方式上述进程间的方式都可沿用，且还有自己独特的几种：

A.互斥量　　B.自旋锁　　C.条件变量 D.读写锁　　E.线程信号

G.全局变量

值得注意的是，线程间通信用的信号不能采用进程间的信号，因为信号是基于进程为单位的，而线程是共属于同一进程空间的。故而要采用线程信号。综上，进程间通信手段有 8 种。线程间通信手段有 13 种。

此外，进程间采用的通信方式要么需要切换内核上下文，要么要与外设访问(有名管道，文件)。所以速度会比较慢。而线程采用自己特有的通信方式的话，基本都在自己的进程空间内完成，不存在切换，所以通信速度会较快。也就是说，进程间与线程间分别采用的通信方式，除了种类的区别外，还有速度上的区别。另外，进程与线程之间穿插通信的方式，除信号以外其他进程间通信方式都可采用。

5).控制方式的异同

进程与线程的身份标示 ID 管理方式不一样，进程的 ID 为 pid_t 类型，实际为一个 int 型的变量(也就是说是有限的)：

/usr/include/unistd.h:260:typedef __pid_t　pid_t;

/usr/include/bits/types.h:126:# define __STD_TYPE　typedef

/usr/include/bits/types.h:142:__STD_TYPE　__PID_T_TYPE　__pid_t;

/usr/include/bits/typesizes.h:53:#define __PID_T_TYPE　__S32_TYPE

/usr/include/bits/types.h:100:#define　__S32_TYPE　　int

在全系统中，进程 ID 是唯一标识，对于进程的管理都是通过 PID 来实现的。每创建一个进程，内核去中就会创建一个结构体来存储该进程的全部信息：

注：下述代码来自 Linux 内核 3.18.1

include/linux/sched.h:1235:struct task_struct {

```
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
...
    pid_t pid;
    pid_t tgid;
...
};
```

每一个存储进程信息的节点也都保存着自己的 PID。需要管理该进程时就通过这个 ID 来实现(比如发送信号)。当子进程结束要回收时(子进程调用 exit()退出或代码执行完)，需要通过 wait()系统调用来进行，未回收的消亡进程会成为僵尸进程，其进程实体已经不复存在，但会虚占 PID 资源，因此回收是有必要的。

线程的 ID 是一个 long 型变量：

/usr/include/bits/pthreadtypes.h:60:typedef unsigned long int pthread_t;

它的范围大得多，管理方式也不一样。线程 ID 一般在本进程空间内作用就可以了，当然系统在管理线程时也需要记录其信息。其方式是，在内核创建一个内核态线程与之对应，也就是说每一个用户创建的线程都有一个内核态线程对应。但这种对应关系不是一对一，而是多对一的关系，也就是一个内核态线程可以对应着多个用户级线程。

还是请读者参看《Linux 线程的实质》普及相关概念。此处贴出 blog 地址：http://my.oschina.net/cnyinlinux/blog/367910

对于线程而言，若要主动终止需要调用 pthread_exit()，主线程需要调用 pthread_join()来回收(前提是该线程没有被 detached，相关概念请查阅线程的“分离属性”)。像线发送线程信号也是通过线程 ID 实现的。

6).资源管理方式的异同

进程本身是资源分配的基本单位，因而它的资源都是独立的，如果有多进程间的共享资源，就要用到进程间的通信方式了，比如共享内存。共享数据就放在共享内存去，大家都可以访问，为保证数据写入的安全，加上信号量一同使用。一般而言，共享内存都是和信号量一起使用。消息队列则不同，由于消息的收发是原子操作，因而自动实现了互斥，单独使用就是安全的。

线程间要使用共享资源不需要用共享内存，直接使用全局变量即可，或者 malloc()动态

申请内存。显得方便直接。而且互斥使用的是同一进程空间内的互斥量，所以效率上也有优势。

实际中，为了使程序内资源充分规整，也都采用共享内存来存储核心数据。不管进程还是线程，都采用这种方式。原因之一就是，共享内存是脱离进程的资源，如果进程发生意外终止的话，共享内存可以独立存在不会被回收(是否回收由用户编程实现)。进程的空间在进程崩溃的那一刻也被系统回收了。虽然有 coredump 机制，但也只能是有限的弥补。共享内存在进程 down 之后还完整保存，这样可以拿来分析程序的故障原因。同时，运行的宝贵数据没有丢失，程序重启之后还能继续处理之前未完成的任务，这也是采用共享内存的又一大好处。总结之，进程间的通信方式都是脱离于进程本身存在的，是全系统都可见的。这样一来，进程的单点故障并不会损毁数据，当然这不一定全是优点。比如，进程崩溃前对信号量加锁，崩溃后重启，然后再次进入运行状态，此时直接进行加锁，可能造成死锁，程序再也无法继续运转。再比如，共享内存是全系统可见的，如果你的进程资源被他人误读误写，后果肯定也是你不想要的。所以，各有利弊，关键在于程序设计时如何考量，技术上如何规避。这说起来又是编程技巧和经验的事情了。

7).个体间辈分关系的迥异

进程的备份关系森严，在父进程没有结束前，所有的子进程都尊从父子关系，也就是说 A 创建了 B，则 A 与 B 是父子关系，B 又创建了 C，则 B 与 C 也是父子关系，A 与 C 构成爷孙关系，也就是说 C 是 A 的孙子进程。在系统上使用 pstree 命令打印进程树，可以清晰看到备份关系。

多线程间的关系没有那么严格，不管是父线程还是子线程创建了新的线程，都是共享父线程的资源，所以，都可以说是父线程的子线程，也就是只存在一个父线程，其余线程都是父线程的子线程。

8).进程池与线程池的技术实现差别

我们都知道，进程和线程的创建时需要时间的，并且系统所能承受的进程和线程数也是有上限的，这样一来，如果业务在运行中需要动态创建子进程或线程时，系统无法承受不能立即创建的话，必然影响业务。综上，聪明的程序员发明了一种新方法——池。

在程序启动时，就预先创建一些子进程或线程，这样在需要用时直接使唤。这就是老人口中的“多生孩子多种树”。程序才开始运行，没有那么多的服务请求，必然大量的进程或线

程空闲，这时候一般让他们“冬眠”，这样不耗资源，要不然一大堆孩子的口食也是个负担啊。对于进程和线程而言，方式是不一样的。另外，当你有了任务，要分配给那些孩子的时候，手段也不一样。下面就分别来解说。

进程池

首先创建了一批进程，就得管理，也就是你得分开保存进程 ID，可以用数组，也可用链表。建议用数组，这样可以实现常数内找到某个线程，而且既然做了进程池，就预先估计好了生产多少进程合适，一般也不会再动态延展。就算要动态延展，也能预估范围，提前做一个足够大的数组。不为别的，就是为了快速响应。本来做进程池的目的也是为了效率。

接下来就要让闲置进程冬眠了，可以让他们 pause()挂起，也可用信号量挂起，还可以用 IPC 阻塞，方法很多，分析各自优缺点根据实际情况采用就是了。然后是分配任务了，当你有任务的时候就要让他干活了。唤醒了进程，让它从哪儿开始干呢？肯定得用到进程间通信了，比如信号唤醒它，然后让它在预先指定的地方去读取任务，可以用函数指针来实现，要让它干什么，就在约定的地方设置代码段指针。这也只是告诉了它怎么干，还没说干什么(数据条件)，再通过共享内存把要处理的数据设置好，这也子进程就知道怎么做了。干完之后再来一次进程间通信然后自己继续冬眠，父进程就知道孩子干完了，收割成果。最后结束时回收子进程，向各进程发送信号唤醒，改变激活状态让其主动结束，然后逐个 wait()就可以了。

线程池

线程池的思想与上述类似，只是它更为轻量级，所以调度起来不用等待额外的资源。要让线程阻塞，用条件变量就是了，需要干活的时候父线程改变条件，子线程就被激活。线程间通信方式就不用赘述了，不用繁琐的通信就能达成，比起进程间效率要高一些。线程干完之后自己再改变条件，这样父线程也就知道该收割成果了。整个程序结束时，逐个改变条件并改变激活状态让子线程结束，最后逐个回收即可。